Reliable Communication in an Unreliable Environment

Kenneth P. Birman
Thomas A. Joseph

TR 85-694
July 1985
Revised September 1985

TECHNICAL REPORT

Department of Computer Science
Cornell University
Ithaca, New York

12

Reliable Communication in an Unreliable Environment[1]

Kenneth P. Birman
Thomas A. Joseph

TR 85-694
July 1985
(Revised September 1985.)

Department of Computer Science
Cornell University
Ithaca, NY 14853

[1] This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 5378, Contract DA903-85-C-0124, and by the National Science Foundation under grant DCR-8412582. The views, opinions and findings contained in this report are those of the author and should not be construed as an official Department of Defense position, policy, or decision.

Reliable Communication in an Unreliable Environment

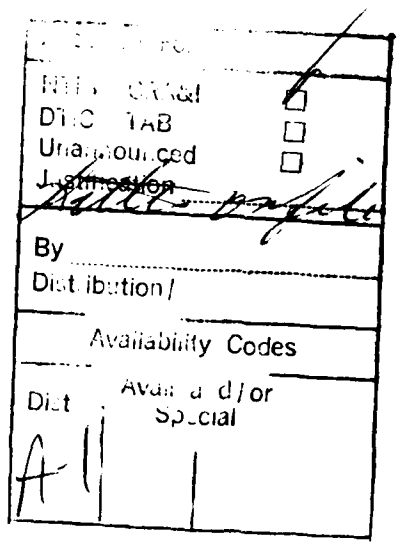
Kenneth P. Birman and Thomas A. Joseph[1]

Department of Computer Science
Cornell University, Ithaca, New York

ABSTRACT

We report on the design and correctness of a communication facility for a distributed computer system. The facility provides a variety of broadcast protocols, which are used to transmit messages reliably to sets of destination processes. These protocols attain high levels of concurrency while respecting application-specific ordering constraints. They also ensure that processes observe consistent orderings of events, including process failures and recoveries. A review of several uses for the protocols in a large fault-tolerant program illustrates the simplification of higher-level algorithms made possible by our approach.

TECHNICAL REPORT TR-85-694



[1]This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 5378, Contract MDA903-85-C-0124, and by the National Science Foundation under grant DCR-8412582. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

1. Introduction

This paper presents a set of communication primitives for supporting distributed computations in an environment where failures could occur. We are primarily concerned with *halting* failures, whereby a process stops executing without performing any incorrect actions. Each distributed computation is represented as a set of events operating on a process-state and a partial order on those events, corresponding to the thread of control. The types of events considered include local computations by a process, broadcasts from a process to a set of processes, broadcasts subject to predetermined ordering constraints, process failures, and recoveries.

Our premise is that event orderings should be subsumed into the communication layer of a distributed system. In addition, since increasing concurrency generally improves performance in distributed systems, we ask how much communication-level concurrency can be achieved while still respecting event ordering constraints specified by the computations. An important feature of our approach is that it enables a process to make assumptions about the event orderings that will be observed by other processes in the system. This simplifies higher-level code by making event orderings more predictable, and permits distributed computations to be implemented with reduced risk of inconsistent actions being taken.

An example will illustrate the class of problems that are addressed here. Consider a process p that is updating a replicated data item maintained by a set of *data managers*. Assume that this update is performed using a *reliable broadcast:* if any data manager receives the broadcast and remains operational, all data managers will receive it. If p fails, a data manager could observe any of several outcomes:

1. The data manager receives the update and then detects the failure.
2. It detects the failure and receives the update later.
3. It detects the failure and the update is not delivered (anywhere).

It may be difficult for a data manager to distinguish cases 2 and 3. Moreover, if some data managers experience the first outcome and others the second one, the system must still behave

correctly. One way to address problems such as these is for each process to run an agreement protocol to decide on what action to take after it detects a failure [Skeen-a]. This approach could be slow because it is synchronous, and expensive because each process has to run such a protocol. Another possibility is to discard messages that are received by a process after it has learned that the sender has failed. However, inconsistencies may arise if messages are discarded by one process but retained by another one that learns of the failure later. A third alternative, representative of the general approach of this paper, is to construct a broadcast protocol that orders messages relative to failure and recovery events such that these problems do not arise. Using the primitives we develop here, a data manager can perform an update immediately upon receiving the corresponding message, because it is guaranteed that all other data managers will receive the message. It can take a recovery action immediately after detecting a failure, because no other data manager will observe an inconsistent ordering of events. The primitives also ensure that every data manager experiences the same sequence of events; this makes programming a computation that performs such distributed updates easier.

The remainder of the paper is structured as follows. Section 2 discusses the presumed environment in more detail. Section 3 describes the communication primitives, and section 4 gives protocols to implement them in a local network. Finally, section 5 applies the primitives to a fault-tolerant system that we have implemented at Cornell.

2. System model

A distributed system is modelled as a collection of processes possessing local states and communicating by messages. Processes do not share memory or maintain synchronized clocks. The term *failure* denotes a *halting* failure: a process ceases execution without taking any (visible) incorrect or malicious actions [Schlichting]. No information survives a failure (by fault-tolerance we refer to continued operation in the presense of failures, not recovery from "stable" storage). If the site at which a failed process was executing remains operational, we assume that the failure is detected (e.g. by the operating system) and that any interested parties are notified. On the other

hand, if a sites crashes, all the processes executing on it fail, and processes at other sites can only detect this by timeouts. The communication system can also fail: it can lose and duplicate messages, or deliver them out of order. Our protocols block, but do not take erroneous actions, if the system *partitions* into subgroups of sites within which communication remains possible but between which it is degraded or impossible.

Clearly, failure detection by timeout cannot be more reliable than the underlying communication system: a series of message losses can always mimic a failure. Moreover, the order in which failures are perceived to have occurred may vary from process to process. These observations lead us to adopt a *logical* approach to failure handling, rather than a *physical* one. That is, instead of a process acting directly after it detects a failure, which could lead to inconsistent actions, a protocol is run to reach agreement with other processes that a failure event has occurred, and to order it with respect to other events. This is meaningful because we have the freedom to pretend that events like message delivery took place either before or after the failure, provided that no evidence to the contrary survived it. The basic property of a logical failure is that after a process learns of such an event or observes the relative ordering of such events, it will never communicate with another process whose state is inconsistent with this information.

3. The communication primitives

This section first defines two broadcast primitives, *BCAST* and *OBCAST*, and describes their behavior in the absence of failures. Failures and recoveries are then included by treating them as a third type of broadcast, *GBCAST*. All the broadcast primitives are *atomic*, that is, a broadcast made to a set of processes is either received[2] by all operational processes, or by none at all, even in the presence of failures. Each broadcast B has a unique identifier, which we denote as $ID(B)$. The process that initiates a broadcast B is denoted $SENDER(B)$, while the set of processes to which B is sent is denoted $DESTS(B)$.

[2]Reception is understood to include the *indirect* observation of a message. For example, if process p receives a message m, communicates with process q, and then fails, the state of q may reflect the contents of m. To keep the sys-

3.1. The *BCAST* primitive

Consider a set of processes that maintain copies of a replicated data structure that represents a queue. If items are inserted into and removed from each copy of the queue in the same order, no inconsistencies will arise among copies. The *BCAST* primitive is provided for applications such as this, where the order in which data are received at a destination must be consistent with the order at other destinations, even though this order is not determined in advance. A *BCAST* is invoked as *BCAST*(*msg*, *label*, *dests*), where *msg* is the message to be broadcast, *label* is a string of characters, and *dests* is the set of processes to which the message must be delivered. *BCAST*'s are atomic: every operational destination receives *msg*, or none does. In addition, if two *BCAST*'s with the same label have destinations in common, they will be delivered in the same order at all overlapping destinations. The replicated queue described above can thus be implemented by using *BCAST*'s to broadcast insert or delete instructions to the various copies, using a queue-id for the *BCAST* label.

3.2. The *OBCAST* primitive

For some applications, it is not sufficient that broadcasts are received in the same order at overlapping destinations - it is also necessary that this order be the same as some predetermined one. On the other hand, a consistent delivery order for messages originating in unrelated senders may be less important. As an example, consider a computation that first sets copies of a replicated variable to zero and later increments the variable. Here, it is not enough for the two operations be carried out in the same order at all copies - the increment must always occur second. However, if independent computations were to access such a replicated variable, some other method would normally be used to synchronize the accesses, making it unlikely that the both would broadcast updates concurrently. The *ordered broadcast* primitive, *OBCAST(msg, olabel, dests)* is used to enforce a delivery ordering desired by a single sender process, but with minimal synchronization. Here, *olabel* is a label that can be compared with other *olabels* using a system-

tem state consistent, unless process q also fails, m must be delivered to all its destinations.

wide algorithm, to yield a partial order on *OBCAST*'s. We write $olabel_1 \leftarrow olabel_2$, if $olabel_1$ and $olabel_2$ are comparable, and $olabel_1$ is less than $olabel_2$. Note that we allow for *olabels* to be incomparable, that is for neither $olabel_1 \leftarrow olabel_2$ nor $olabel_2 \leftarrow olabel_1$ to hold. We use $OLABEL(B)$ to represent the *olabel* of broadcast B, and for brevity write $B \leftarrow B'$ to mean $OLABEL(B) \leftarrow OLABEL(B')$. An application uses *olabel*'s to indicate the order in which broadcasts should be delivered.

What constraints do *olabel*'s place on the order of broadcast deliveries? Some orderings specified by *olabels* are trivially satisfied. For example, if two *OBCAST*'s have no destinations in common, there is no actual constraint on the order of message delivery, regardless of how their labels may compare. On the other hand, some specifiable orderings are unenforceable. An *OBCAST* with an *olabel* less than one that has already been delivered clearly cannot be delivered in the desired order. This calls for a restriction on allowable *olabels*. Fortunately, most applications require an order to be enforced between two broadcasts only if the outcome of one could causally affect the other. The notion of *potential causality* in an asynchronous distributed system in which information is exchanged only by transmitting messages is studied in [Lamport]. In such a system, a broadcast B is said to be *potentially causally related* to a broadcast B' only if they were sent by the same process and B' occurred after B, or if B was delivered at $SENDER(B')$ before B' was sent (or there is a chain of such receivers and senders linking B to B'). We restrict labels on *OBCASTS* to disallow $OLABEL(B')$ from being less than that of $OLABEL(B)$ if they have the same sender and B' is sent after B, or if B was delivered at $SENDER(B')$ before B' was sent[3]. This is not a major restriction because such orderings cannot be enforced unless the system has knowledge of which future broadcasts a broadcast must wait for. Note, however, that in general *OBCAST* will not provide the sort of consistent delivery orderings given by *BCAST*: it satisfies a weaker ordering constraint.

[3] More accurately, if a broadcast *is* labeled in this way, the *OBCAST* primitive does not guarantee that this order will be observed.

A broadcast primitive could be designed that orders any two broadcasts that are potentially causally related. This is stronger than necessary, however. Consider a broadcast B made by a process p to update copies of a replicated variable x. Let this be followed by a broadcast B' by p to update copies of y. Even though there is a potential causal relation between B and B' because B' occurred after B, there may be no real causal relation between them. In such cases, there would be no reason to order the delivery of B before that of B'. Unecessarily ordering such broadcasts is inefficient because it limits the possible concurrency in the system. The *OBCAST* primitive uses *olabels* to identify which causal relationships are significant and should be observed. Essentially, it orders broadcasts relative to each other if they are potentially causal *and* if the *olabels* indicate that the potential causal relationships are significant.

We now formally define the ordering properties of *OBCAST*'s. Given $\leadsto$ as above, let the relation *precedes* between *OBCAST*'s be the transitive closure of the following two relations:

A. B *precedes* B' if $B \leadsto B'$ and the same process p sends B before it sends B'.

B. B *precedes* B' if $B \leadsto B'$ and B is delivered at $SENDER(B')$ before B' is sent.

Then *OBCAST*'s have the following properties. They are atomic, and if B *precedes* B', then B is delivered before B' at any overlapping destination.

The *OBCAST* primitive may seem to be too weak because it cannot enforce orderings that may be desired between broadcasts that are not potentially causal. Consider a process p that instructs a set of devices, "place wine bottles under taps," and a process q that orders, "open taps". Clearly, it is desirable that the first broadcast be delivered everywhere before the second. However, in an asynchronous system in which there is no upper bound on message delivery times, the only way this can be implemented without wasting a lot of wine is to require that the devices send q a message when the wine bottles have indeed been placed under the tap. These messages causally relate the broadcast from p to that from q, and *OBCAST*'s can then be used to enforce the desired ordering. In general there will be little or no occasion to order asynchronous broadcasts that are not potentially causal. Thus the *OBCAST* is strong enough for most applications.

Note that the accuracy with which *olabel*'s represent the dependency between broadcasts could limit concurrency: if *B precedes B'*, *OBCAST* will deliver *B* first even if the semantics of *B* and *B'* are such that they are actually independent.

3.3. Broadcasts in the presence of failures and recoveries

In fault-tolerant systems, it is frequently necessary for the members of a group of processes to be able to monitor the status of one another. They can then take actions based on failures or recoveries of group members. As an example, consider a fault-tolerant server that is implemented using a group of processes as follows. A request for the service is broadcast to all the members of the group. The operational process having the smallest ID responds to the request. For this implementation to function correctly, it is necessary that all the members of the group have the same view of which members are operational. Otherwise no member may respond (as may happen if all operational members believe that a failed process with a smaller ID is still operational), or more than one member may do so (if an operational member believes that a process with a smaller ID has failed when it has not). Further, if there has been a change in the status (operational / failed) of a member, it is necessary for all the processes to agree on whether a request should be handled before or after the change in status, so that they may consistently decide on which process should respond to the request. Although these problems could be addressed by running an agreement protocol each time a failure or recovery is suspected, and / or by executing a consensus protocol before responding to any request, it would be expensive and complex to do so. A simpler method, described below, is to provide a *process group* abstraction having the property that changes in the group membership (including failures and recoveries) are ordered with respect to ongoing broadcasts.

In our system, each process p is initially in a process group G_p containing only itself. A process p can join or withdraw from any process group G using the primitive *GBCAST*(*action*, p, G), where *action* is either *join* or *withdraw*. Each member of a process group maintains a *process group view*, giving the current membership of the group. Invoking *GBCAST* results in a message

being broadcast to all the members of the group informing them of the action. Upon receipt, the process group view of each member is updated accordingly. *GBCAST*'s are atomic and are ordered in the same way relative to all *BCAST*'s and *OBCAST*'s at the destinations. In particular, it is not possible for a *BCAST* or an *OBCAST* to be received *before* a *GBCAST* by some of the members of a process group and *after* the same *GBCAST* by other members. The consequence of this is that a member of a process group can respond to any *BCAST* or *OBCAST* with the guarantee that any other member will respond to it based on the same process group view, without needing to carry out an agreement protocol to ensure this.

Failure decisions are ordered with respect to ongoing broadcasts by simulating a GBCAST(*p_has_failed*, *p*,*G*) from a failed process *p* to all the process groups *G* to which it belonged (this is called a *failure GBCAST*). If the failure is an isolated one, the *GBCAST* can be issued by a supervisory process at the site where the failure occurred. If a site crashes, then the software handling failure detection (Sec. 4.1) initiates *GBCAST*'s for every process at the failed site. Sites receiving a failure *GBCAST* remove the failed process from their process group view, as for a *withdraw*. When a process recovers, it *GBCAST*s its intention to join process groups. Thus failures and recoveries appear as simple changes in the membership of process groups. An additional property is provided for failure *GBCAST*'s: they are delivered *after* any *BCAST* or *OBCAST* from the failed process. Thus after a process has learned of the failure of another process, it is guaranteed to receive no more messages from that process.

If the fault-tolerant server described above is implemented using a process group, each member can independently decide whether or not to respond to a request based on its process group view. The ordering properties of *GBCAST*'s ensure that inconsistencies do not arise. Moreover, *since all members observe the same sequence of view transitions, members can react consistently to* failures and recoveries. The process group abstraction thus greatly simplifies the construction of fault-tolerant software.

3.4. Flush primitive

In certain situations, a process needs to know that a message has been actually received at its destinations before it can continue. For example, consider a process that broadcasts a checkpoint to a set of backup processes. If it fails while the broadcast is still in progress, the broadcast might not be delivered to any backup (cf. definition of *atomicity*), and the failure handling action would not occur. One way to address this is for a sender to request acknowledgements from the destinations, and to wait until the acknowledgements are received. Instead of having to do this explicitly each time, a **flush** primitive is provided. A process calling **flush** is blocked until all its pending *OBCAST*'s have actually been delivered, and is then allowed to continue.

3.5. Group addressing

All of our protocols require that a sender explicitly name the set of destination processes for each broadcast. A problem arises if a sender wishes to broadcast to *all* the members of a process group. If the group grows after the broadcast is initiated but before it is delivered, any new members would not receive it. A way to resolve this is for each process group member to number its process group views sequentially. Any process can then *cache* (possibly out of date) process group membership information and view numbers for groups with which it communicates. To transmit a *BCAST*, *OBCAST*, or *GBCAST* to all members of a group G, the cached information would be used to compute $DESTS(B)$, and the view number included in the message. On delivery, if a recipient finds that the process group view has changed, it rejects the message. Since all recipients have the same view when they receive the message, they all reject it if any does so. A rejected broadcast can then be retransmitted to an updated set of destinations, and the cache updated.

Some care is needed when updating the cache, to ensure that the *OBCAST* delivery order is preserved. In particular, consider three *OBCAST*'s $A \rightarrow B \rightarrow C$, and assume that A and B have been transmitted using incorrect destinations. If the cache is updated promptly after A is rejected, C could be transmitted using the corrected destinations before B is rejected and retransmitted. It

will now appear that the *B* and *C* are not causally ordered, and hence *C* might be delivered first. This problem is avoided by invoking **flush** before changing the contents of a cache.

3.6. Related work

The *BCAST* primitive described above is similar to *atomic broadcast* [Chang], where the problem is to send messages one site to all other sites in a network, with the same reception order everywhere. Chang does not define atomic broadcast on a process-process basis, nor are other types of broadcast considered. An interesting comparison can be drawn between this work and that reported in [Christian], where a class of atomic broadcast protocols are developed, under varying assumptions about the environment. The atomicity property addressed in that paper is essentially the same as in the *BCAST* protocol given here. However, whereas our *OBCAST* protocol weakens the ordering property (we still call it atomic), in [Christian] a strong ordering constraint is taken as be part of the definition of atomicity (recall that *OBCAST* may not deliver messages from unrelated processes in consistent orders at overlapping destinations). In practice, we have found *OBCAST* to be valuable when building a system that manages replicated data, because it relaxes the degree of synchronization while still ensuring that processes that survive a failure are left in consistent states. In addition, our use of *GBCAST* to maintain process groups is new, although the idea of grouping processes together is not. For example, CIRCUS supports process *troupes*, but in an environment subject to assumptions that simplify the broadcast ordering problem [Cooper]. Specifically, process executions are deterministic (this is not required in our work), and if a troupe receives messages from independent sources, the message delivery order must be the same at all members. The ADAPLEX system supports a protocol, *exclude*, which is used to order replicated updates in a database system with respect to failure [Goodman], much like *GBCAST* is ordered with respect to other broadcast types.

4. Implementation of the communication abstraction

This section gives implementations for the communication abstraction, targeted to a collection of computers interconnected by a local network. First, the "raw" environment is transformed into one satisfying the desired failure and communication properties. Next, the *BCAST*, *OBCAST* and *GBCAST* protocols are given. Finally, a garbage collection mechanism is described. Figure 1 illustrates the overall system structure.

4.1. The inter-site layer

The inter-site communication layer converts halting failures and admissible communication failures (message loss, delayed delivery, and out-of-order delivery) into a *site-view* abstraction,

---

Distributed computations
[BCAST / OBCAST] | [GBCAST]
Associative store
View manager
Inter-site protocol: send, status, hello
Unreliable packet transport, e.g. TCP/IP, PUP, etc.

Figure 1: Layered structure of the communication subsystem.

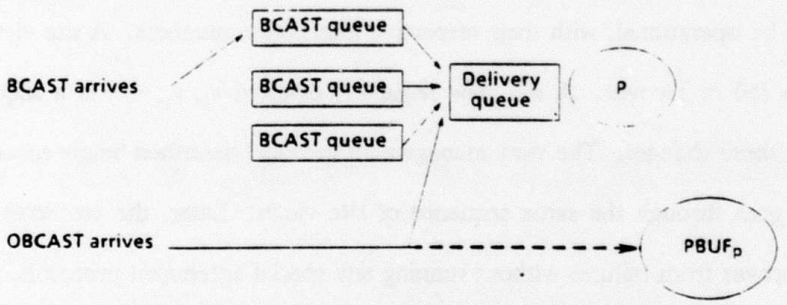


Figure 2: Data structures used for BCAST and OBCAST

---

defined below. The layer provides two primitives: **send(m, dest)** for sending message *m* to site *dest*, and **status(m)**, which returns *sent* if the destination has acknowledged receipt of the message or if a failure protocol has been started for the destination site, as described below. Intuitively, a message has been *sent* if the future behavior of the system will be consistent with the message having already been delivered.

The intersite layer employs a windowed acknowledgement protocol for ordered, loss-less site-to-site message transmission. To detect failures, each site sends a "hello" message to all other sites periodically; if a hello message is not received from a site within a reasonable period, it is assumed to have failed, triggering the change of view protocol. If a site is slow to send messages, it may therefore be forced to undergo recovery (the probability of error can be made small by picking a large timeout interval or introducing a protocol phase to allow other sites to prevent execution of the failure algorithm). A site *incarnation* number is incremented each time a site recovers; henceforth, the term "site" always means "incarnation of a site." Messages from a failed incarnation are discarded, and a *you are dead* message is returned to the sender. Messages addressed to a different incarnation than the current one are discarded.

4.2. View management

The view management layer ensures that each site in the system has a consistent picture of site failures and recoveries occurring in the system. Each site has a *site view*, which is the set of sites it deems to be operational, with their respective incarnation numbers. A site view is changed when other sites fail or recover. A *site view sequence*, denoted $v_0, v_1, \cdots$ is a sequence of site views, reflecting these changes. The view management protocol described below ensures that each operational site goes through the same sequence of site views. Later, the protocols take advantage of this to recover from failures without running any special agreement protocols.

Each site maintains a copy of the site view sequence, initialized in some consistent way when the system cold-starts. We assume that the sites in a view can be ordered uniquely according to the view in which they were first operational, with ties broken by site-id. The "oldest" site in this

order is called the *view manager*, and is responsible for initiating the view management protocol when it detects a site failure or recovery. If a site determines that all sites older than itself have failed, it takes over as the new view manager. Note that the sequence of view managers is a stable property: extensions to the view sequence extend the sequence of managers without changing the subsequence on which sites have already agreed.

The view management protocol is based on a two phase commit protocol. Let $v_0, v_1, \cdots, v_t$ be the current site view sequence.

1. On detecting failures or recoveries, the view manager computes a proposed *view extension* $v_{t+1}, v_{t+2}, \cdots, v_{t+k}$. (If no failures occur during the execution of the protocol, the length of the extension is 1, that is, v_{t+1} contains all the changes to the current site view. Failures occurring during the execution of the protocol may cause the site view sequence to be extended by more than one view, as described below.) It ceases to accept messages from site-incarnations not in v_{t+k} and sends the proposed view extension to the sites in v_{t+k}.
2. On receiving a proposed view extension, a site first ceases to accept messages from site-incarnations not in v_{t+k}.
   a. If the site has not previously received a proposed extension, or the new one includes all the changes (failures and recoveries) recorded in the old one, the site saves the new proposed extension. Then, it replies to the view manager with a *positive acknowledgement*.
   b. Otherwise, the site has previously received a proposed view extension recording events that are not included in the new one. It replies with a *negative acknowledgement*, giving the events that were missing.
3. The view manager collects acknowledgements.
   a. If all the acknowledgements were positive, it sends a *commit message* for the proposed extension.
   b.

If additional failures or recoveries have been detected, or negative acknowledgements were received, the view manager updates its proposed extension and re-executes from step 1.

If the view manager fails, a new site takes over as view manager, and proceeds as follows:

4. If this new view manager has an uncommitted view extension, the previous view manager may have sent some commit messages before failing. It appends a new site view containing the failure of the old view manager to its pending extension and starts the protocol from step 1.
5. If the new view manager has received a committed extension, and has no pending one, it must assume that some sites did not receive the commit. It appends a new view to the most recently committed extension and continues from step 1. Participants ignore a committed prefix of a proposed extension.

To establish the correctness of the protocol, consider the cases that can arise:

1. If the view manager doesn't fail, all sites obtain the same committed view extensions.
2. If the view manager fails and any site has a committed view extension, than all sites have acknowledged that extension. The new view manager will eventually commit the extension everywhere.
3. If the view manager fails after it has distributed a proposed extension to a subset of sites, and that proposed extension is not known to the new view manager, then any site knowing the extension will send a negative acknowledgement to the new coordinator when the protocol is restarted, and the coordinator will then distribute it during an additional protocol phase.

The following issues arise because sites may detect failures and recoveries of other sites at different times and in arbitrary order. First, the order in which view managers commit site views becomes the order accepted by the system, even if individual sites may have detected failures and recoveries in a different order. Second, a view manager may erroneously decide that a site has failed (because it is slow to respond). In this case, all sites consider the site in question to have

failed[4], and respond to any message from it with a "you are dead" message. Such a site is said to be *killed*, as it is forced to undergo recovery with a new incarnation number. Third, it is possible for a site a to believe that a site b has failed, for b to believe that a has failed, and for each of them to consider themselves as the view manager. In this situation, one or both will be killed, otherwise some site would have to acknowledge two contradictory views from two different view managers, which cannot happen.

Two final problems remain to be addressed. If it is desired that the system be able to recover if all sites fail, a protocol in [Skeen-b] can be run to reconstruct the view sequence from copies saved on non-volatile storage. Also, if network partitioning can occur, erroneous actions can be prevented by requiring that sites cease to operate if number of operational sites in a view drops below a quorum. □

4.3. The protocols

This section gives implementations for *BCAST*, *OBCAST*, and *GBCAST*, deferring garbage collection issues to Sec. 4.3. The broadcast protocols order messages addressed to a process as necessary, and place them on the *delivery queue* for the process. A process removes messages from its delivery queue in FIFO order. The protocols use other queues in which they buffer messages before placing them on delivery queues. Figure 2 illustrates the relationship between the various queues.

4.3.1. BCAST protocol

Our *BCAST* protocol is based on a two-phase protocol by [Skeen-c]. The protocol maintains a set of priority queues for each process, one for each *BCAST* label, in which it buffers messages before placing them on the delivery queue. We assume that priority values are integers, with a process-ID appended as a suffix to disambiguate between priorities assigned by different processes. Each message in the buffers is tagged *deliverable* or *undeliverable*. The protocol to

[4]This is true unless the network becomes *partitioned*, that is, a group of sites continues to remain operational, but unable to communicate with the other sites. Network partitioning is considered later.

implement *BCAST*(*msg*, *label*, *dests*) is as follows:

1. The sender transmits *msg* to its destinations, using **send**.
2. Each recipient adds the message to the priority queue associated with *label*, tagging it as *undeliverable*. It assigns this message a priority larger than the priority of any message that was placed in the queue, with the process-id of the recipient as a suffix. It then informs the sender of the priority it assigned to the message.
3. The sender collects responses from recipients that remain operational. It then computes the maximum value of all the priorities it received, and sends this value back to the recipients.
4. The recipients change the priority of the message to the value they receive from the sender, tag the message as *deliverable*, and resort their priority queues. They then transfer messages from the priority queue to the delivery queue in order of increasing priority, until the priority queue becomes empty or the message with the lowest priority is undeliverable. In the latter case, no more messages are transferred until the message at the head of the queue becomes deliverable.

If a failure occurs, any site that has a message tagged *undeliverable* from a failed sender can detect this by observing a change in view, and can take over as the new coordinator to complete the protocol. It does so by interrogating participants about the status of the message. A participant being interrogated either has never received the message, or responds with the priority and tag. The new coordinator collects responses. If any process had marked the message *deliverable*, the new coordinator distributes the corresponding priority to the other processes (step 3). Otherwise, it resumes from step 1. Note that this scheme requires that each process retain information about messages even after they are placed on the delivery queue; garbage collection is discussed in Sec. 4.4.

Correctness. The protocol is atomic because before any recipient tags a message as *deliverable*, all destinations must have received copies of it. If a failure occurs after that, a destination that has a copy tagged *undeliverable* will complete the protocol. Thus if the message is delivered

at any destination, it will be delivered at all of them.

We now show that every message is delivered in the same order at all overlapping destinations. If the priorities of any two messages were assigned by the same process, they cannot be equal. If they were assigned by different processes, the process-ID that is suffixed can be used to order them, should the priority values be equal. Thus every deliverable message has a *unique* priority assigned to it. Messages addressed to overlapping destinations are delivered everywhere in this order. Note that it is not possible for the priority of an undeliverable message to be changed to become less than that of one that has already been delivered. The final priority is the maximum of all assigned priorities; thus a message can only be moved later in a priority queue. □

4.3.2. OBCAST protocol

Our *OBCAST* protocol operates by ensuring that whenever a message B is sent from a process p to a process q, a copy of every undelivered message B' that *precedes* B is also sent to q with B, even if q is not a destination for B'. Thus a message may travel from process to process before it reaches a destination, and multiple copies could be delivered by different routes (duplicates are discarded). It follows that if a message B is delivered to a process p, then copies of all messages addressed to p that *precede* B also arrive with B, or have arrived earlier. Messages addressed to p can therefore be delivered in order. We first describe a simple but inefficient *OBCAST* implementation, then show how its efficiency may be improved.

For each process p, there is a message buffer BUF_p, which contains copies of messages sent to and from p, as well as copies of messages that arrive at p en route to other processes. Every message B in BUF_p has fields $ID(B)$ and $REM\_DESTS(B)$ associated with it. When p performs an *OBCAST*(*msg*, *olabel*, *dests*), the message is placed in BUF_p, and $REM\_DESTS(B)$ is initialized to *dests*. If $p \in REM\_DESTS(B)$, a copy of the message is placed on the delivery queue for p, and p is removed from $REM\_DESTS(B)$. The process p can then continue as if the message has already been sent. Messages in BUF_p are later scheduled for transmission. The decision as to when this occurs can be based on advice from higher level algorithms (a message that requires a response

would be transmitted as soon as possible to minimize waiting time), or on factors like the load on the network. We assume only that all messages are scheduled for transmission within finite time. For now, we also assume that a copy of any message placed in BUF_p remains in the buffer indefinitely.

A message B is transmitted from BUF_p at site s to BUF_q at site t as follows:

1. A *transfer packet* $<B_1, B_2, \cdots>$ is first created including all messages B' in BUF_p such that $B' \downarrow B$ and $REM\_DESTS(B')$ is non-empty. The messages are sorted so that if $B_i \downarrow B_j$, then $i < j$.
2. The transfer packet is then transmitted from site s to site t using **send**.
3. When the packet has been *sent*, for each B_i that it contained, q is deleted from $REM\_DESTS(B_i)$ if it was listed there.

When process q receives a packet $<B_1, B_2, \cdots>$, the following is done for each i, in increasing order of i:

4. If $ID(B_i)$ is already associated with a message in BUF_q, then B_i is a duplicate and is discarded.
5. If $q \in REM\_DESTS(B_i)$, B_i is placed on the delivery queue for q, q is removed from $REM\_DESTS(B_i)$, and a copy of B_i is placed in BUF_q.
6. Otherwise, B_i is a message in transit to another process, and it is simply placed in BUF_q.

Correctness. Any process q that receives a message adds a copy of it to BUF_q. Since all messages in BUF_q are scheduled for transmission within finite time, it follows that if any site has received a message and does not fail, the message will eventually be delivered to all the destinations that remain operational. Thus, the protocol is atomic.

To show that messages are delivered in the correct order, it suffices to show that for every pair of messages B and B' delivered to q, if B precedes B', then B is placed on the delivery queue before B'. We first prove that a copy of B will have been placed in $BUF_{SENDER(B')}$ when B' is first placed there. Then any transfer packet that contains B' will also contain B, and B will be ordered

before B' in it. Thus when the first transfer packet containing B' arrives at q, a copy of B will also be received. If B has not arrived in an earlier packet (and hence already placed on the delivery queue), B will now be placed on the delivery queue before B'.

It follows from the definition of the relation *precedes* that if B *precedes* B', there is a sequence of *OBCASTS* $B=B_0, B_1, \cdots, B_n=B'$ such that for all i, $0 < i \leq n$, $B_{i-1} \rightarrow B_i$, $SENDER(B_i) \in REM\_DESTS(B_{i-1})$, and B_{i-1} is received at $SENDER(B_i)$ before B_i is sent. The proof that a copy of B will have been placed in $BUF_{SENDER(B')}$ when B' is first placed there is by induction on n, the length of the shortest sequence satisfying the properties above. If $n = 0$, $B = B'$, and the result follows immediately. Assume that the hypothesis is valid for $n = k$. If $n = k + 1$, consider the messages B and B_k. By the induction hypothesis, a copy of B will have been placed in $BUF_{SENDER(B_k)}$ when B_k is first placed there. Hence, any transfer packet carrying a copy of B_k will also carry a copy of B. We know that $B_k \rightarrow B_{k+1}$, $SENDER(B_{k+1}) \in REM\_DESTS(_k)$, and B_k is received at $SENDER(B_{k+1})$ before B_{k+1} is sent. Hence, a copy of B will arrive at $SENDER(B_{k+1})$, and be placed in $BUF_{SENDER(B_{k+1})}$ before B_{k+1} is delivered. This gives us the required result. □

There are a number of ways in which the protocol above can be optimized:

- Although the protocol was stated in terms of packets sent from process to process, these packets could be combined to form larger inter-site packets. One inter-site packet could suffice to transfer messages from processes at one site to all destination processes at another. The packet reception rules would be amended to deliver all the messages in a packet that have local destinations at once, and to correctly update the associated *PBUF*'s.
- Rather than keeping a copy of a message in the buffer of each process at a site, the buffers could contain only pointers to a common message pool for all processes at the same site.
- To avoid sending a copy of the same message from process p to process q more than once, a field $SENT\_TO(B)$ can be associated with each message B and updated each time a packet con-

taining the message is sent. The packet generation rules can then be further amended to include B in a packet to a site only if it has not already been sent there.

The problem of deleting a message after it has reached all its destinations (*REM_DESTS* becomes empty) is discussed in Section 4.4.

4.3.3. GBCAST protocol

A *GBCAST*(*action*, p, G) must be ordered relative to other *GBCAST*'s to G, as well as relative to *BCAST*'s and *OBCAST*'s. In addition failure *GBCAST*'s must be delivered after every message from the failed process. These aspects are treated as separate parts in the description of the protocol, then optimizations yielding a more efficient implementation are given.

The first part is carried out only for failure *GBCAST*'s, and ensures that all messages from a failed process are ordered before the *GBCAST*. Say that the process that failed is f.

1.1 The process p running the protocol sends a message to *all* processes in the system, informing them of the start of the failure *GBCAST* for f.

1.2 A process q receiving this message schedules for transmission any message B in BUF_q sent by f that has a process in G in *REM_DESTS*(B). It then waits until the status of these messages turns to *sent*.

1.3 If q belongs to G, q waits until all *BCAST*'s from f have become deliverable. This will happen eventually because some process (perhaps q itself) will take over to complete the *BCAST* protocol.

1.4 The process q then sends an acknowledgement to p. This part of the *GBCAST* ends when acknowledgements have been received from all operational processes.

The second part of the protocol is based on the *BCAST* protocol, and orders *GBCAST*'s to the same group relative to one another, and *GBCAST*'s relative to *BCAST*'s.

2.1 The process p distributes the message *action* to the members of the process group G.

2.2 A recipient q places copies of the message on *all BCAST* priority queues, tagging them *undeliverable*. We assume that there is always a (possibly empty) queue for *every* possible *BCAST* label. It assigns it a priority greater than that of any message that has been placed on *any* of the *BCAST* queues, and sends this priority value back to p (all copies receive the same priority).

2.3 After collecting the responses, p sends the maximum of all the values it received to the members of G, which change the priority accordingly and resort their queues. Unlike in the *BCAST* protocol the messages are not tagged *deliverable*. Thus when a *GBCAST* message reaches the head of a *BCAST* priority queue, further delivery of messages from that queue is suspended.

2.3 When the *GBCAST* message reaches the head of *all BCAST* queues, the next part is begun.

The third part orders *GBCAST*'s relative to *OBCAST*'s. We assume that the *OBCAST* protocol is modified to maintain a list $IDlist_p$ for each process p, containing ID's for *OBCAST* messages that have been placed on the delivery queue of p. For now, assume that the list includes the ID's of *all* such messages. The goal of the protocol is for processes in G to agree on a list *before* of *OBCAST* messages ordered before the *GBCAST*, and to deliver messages accordingly. The third phase executes as follows.

3.1 The process p initiating the protocol contacts all members of G.

3.2 A participant q establishes a FIFO *wait queue* (unless one already exists). Until the *GBCAST* protocol completes, messages that would have been placed on the delivery queue at q by the *OBCAST* protocols are placed on this queue instead.

3.3 If any message B in $IDlist_q$ is in $PBUF_q$ and the remaining destinations of B include sites in G, q must assume that those sites have not yet received a copy of B. Any such message is scheduled for transmission to the destinations in $REM\_DESTS(B) \cap G$, and q waits until their status changes to *sent*. It then sends $IDlist_q$ to p.

3.4 After collecting these messages, p merges all the lists it received, calling the result *before*. It sends *before* to all participants. When a participant q receives the *before* list, any message that was transmitted during Step 3.3 must have arrived, and is on the wait queue unless it has already been delivered. Similarly, during Step 1.2, all *OBCAST* messages from a failed process were either placed on the wait queue or delivered.

Finally, messages are transferred in order to the delivery queue, and normal delivery resumes:

4.1 Each participant q does the following. For each *OBCAST* B in its wait queue, if B is listed in *before*, or if there is some B' in before and $B \leftarrow B'$, or if the *GBCAST* is for a failure of process f and $SENDER(B) = f$, then B is added to *before*.

4.2 Any messages in the wait queue which are also listed in *before* are now transferred to the delivery queue, preserving their relative order. The *GBCAST* message is then placed on the delivery queue.

4.3 If there are no other *GBCAST* protocols in progress, p appends the contents of the wait queue to the delivery queue and deletes the wait queue.

4.4 The *GBCAST* messages are removed from the heads *BCAST* queues, allowing *BCAST* messages to be delivered.

If a failure occurs, any participant can restart the protocol from the beginning. As with *BCAST*, participants reply using the deliverable priority of the *GBCAST* message if they know it; all other steps of the protocol are idempotent and can be repeated without ill effect.

Correctness. *GBCAST* is atomic because no participant can deliver a *GBCAST* message until all have received it, hence if any delivers it, all can restart the protocol.

GBCAST's to the same process group are ordered in the same way at every member because each *GBCAST* is assigned a unique priority value (Step 2.3), and is delivered in this order.

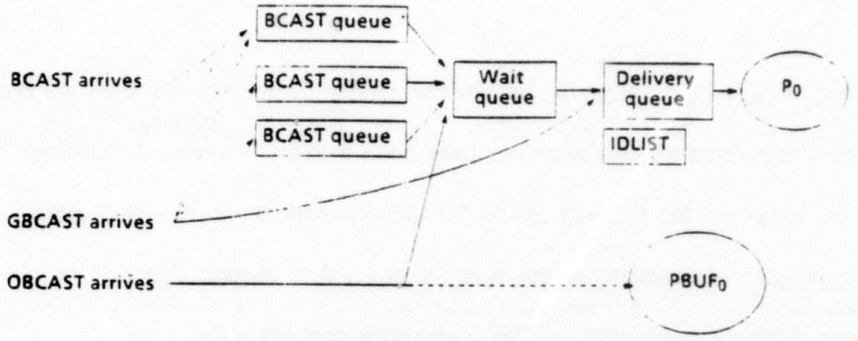


Figure 3: Message flow during GBCAST execution

GBCAST's are ordered consistently with respect to *BCAST*'s because a copy of the message is placed on each *BCAST* queue, and the second part of the *GBCAST* protocol is the same as the *BCAST* protocol.

To show that *GBCAST*'s are ordered in the same way relative to *OBCAST*'s, we must show that if an *OBCAST* is delivered before the *GBCAST* at a member of G, it will be delivered before the *GBCAST* at any other member that it is sent to. The *OBCAST*'s delivered before a *GBCAST* at a process q are those placed in the delivery queue before the wait queue is established, as well as the *OBCAST*'s in the *before* list that are delivered in Step 4.2. Now, any *OBCAST* in the delivery queue before the wait queue is established must be in $IDlist_q$ in Step 3.2 and is hence in the *before* list. Also, any message delivered in Step 4.2 is in the *before* list, or *precedes* some message in the before list. It suffices to show that any message delivered by q arrives in the wait queue of any other destinations in G before Step 4.2 is executed there. This, however, is immediate because a copy of any such message will have been in $PBUF_q$ during Step 3.3, hence q did not respond until it had confirmed their delivery.

Selecting some of the messages from the wait queue to be delivered ahead of others could conceivably upset the *OBCAST* delivery order. But, assume that *OBCAST* B is before B' on some

wait queue, and that B' is delivered during step 4.2 but B is not. Clearly, $\neg B \not\rightarrow B'$, since Step 4.1 would otherwise have listed B in *before*. Thus, the *OBCAST* delivery constraints are respected.

Observe that because of the flush performed during part 1, the protocol does not begin executing until all messages from a failed process f have been delivered to their destinations. Hence, such messages are either on the delivery queue for the destinations or on a wait queue, if some other *GBCAST* protocol was executing at the time. Step 4.1 then ensures that that the *GBCAST* is delivered after any other message from f. This observation completes the proof. □

Optimizations. The *GBCAST* protocol can be optimized simply by merging steps together. Moreover, the flush that is done in part 1 could be invoked directly from the view management protocol - then, instead of doing this on a per-process basis, which would be extremely costly, it would occur on a per-site basis, at relatively low cost. If this were done, a 2 round protocol would result, not counting the cost of the flush, and performance should be acceptable. A method for controlling the length of *IDlist*'s is given below.

4.4. An associative store and distributed garbage collection facility

We now define an *associative store* mechanism, which is used by the above protocols to manage the information associated with message id's. Each site s maintains a local store denoted $STORE_s$. The contents of a store are tuples (*id*,*alist*), where *id* is a broadcast ID and *alist* is a list of zero or more *attributes*. A set of operations are defined on the store for each site (there is no facility for accessing the store at a remote site). The operation **st_add(id)** creates an empty list for the designated ID, **st_insert(id, aname, value)** adds an attribute with name *aname* and value *value* to the list, **st_find(id, aname)** looks up an attribute, and **st_delete(id, aname)** deletes an attribute (but not the ID). The special attribute *DISPOSABLE* is inserted when an entry will no longer be referenced. In the *BCAST* and *GBCAST* protocols, an ID becomes DISPOSABLE at a site running (or completing) the protocol after it transmits commit messages. In the *OBCAST* protocol, an ID becomes DISPOSABLE at a site when the corresponding *REM_DESTS* field is empty.

We now give a method for deleting information associated with a message-id after the ID is marked as *DISPOSABLE* by some site. The method defines a *delete action* which is taken when an ID is discarded; at the end of the section we give delete actions. Since copies of messages may be transmitted to a site while it is running the garbage collection protocol, and message ID's are used to avoid delivery of duplicate copies of messages, some care must be taken to ensure that a copies of a message will not be received after its ID is deleted. Accordingly, the algorithm employs an additional field associated with each message ID in the store, *DONT_SEND*, which is initially null and subsequently lists sites that have run the protocol.

1. Periodically, each site a makes a list of tuples, (*ID*, *DONT_SEND*) for *DISPOSABLE* ID's. It invokes the *delete action* for each listed ID, and then, to each site s, transmits a list $\{ID_i\}$ containing all the *DISPOSABLE* ID_i satisfying $\neg\ s \in DONT\_SEND_i$.

2. On receiving a list $<ID_0, ID_1, \cdots >$, site b takes the following actions.

   a. If ID_i is not already in $STORE_b$, it is added.

   b. The *REM_DESTS* field associated with the ID is made empty and it is marked as *DISPOSABLE*. This ensures that b will not send additional copies of the message and that the ID will eventually be deleted from $STORE_b$.

   c. It adds a to the *DONT_SEND* field associated with this ID in $STORE_b$.

   After processing the list, b sends an acknowledgement to a.

3. After receiving acknowledgements from all operational sites, a deletes the ID, the *DONT_SEND* field, and and other information associated with the ID from $STORE_a$. The *DONT_SEND* field prevents a site from adding an *ID* to its store after deleting it, i.e. when some other site executes the protocol to delete the *ID* from its own store.

BCAST and *GBCAST* have no special delete action -- the priority information that they saved is discarded automatically when the protocol completes. The delete action for *OBCAST* is to remove the message ID from the *idlist* of any processes that have received a copy of it, and to

delete the message itself from $PBUF_p$ for any processes p at the site. Thus, the length of an idlist will be determined by the number of active broadcasts, which should be small.

Correctness. This follows because no site deletes a message-id until all operational sites have sent acknowledgements in step 2, but after step 2.c duplicates of a message will no longer be sent to a site that has run the protocol. □

4.5. OBCAST flush implementation

Flush is invoked in two ways, each having a slightly different implementation. When a process invokes **flush**, the *OBCAST* algorithm is such that if any *OBCAST B* is active, a copy of *B* will be present in $PBUF_p$ for any process p that might take actions causally dependent on the delivery of *B*. Hence, it suffices to schedule all messages in $PBUF_p$ for transmission, and then wait until all have been *sent* to the destinations that remain operational.

If **flush** is invoked for a group address change, a stronger condition is needed, namely that there is no active *OBCAST* that could still be rejected. This is satisfied by doing an *OBCAST* requesting that group members return an acknowledgement. If this *OBCAST* is ordered after all messages that have been sent previously, the acknowledgement will not be received until the messages in question have all been accepted.

5. Applications

The communication subsystem proposed here is being implemented in the *ISIS* system, under development at Cornell. *ISIS* is a distributed computing system that transforms non-distributed abstract type specifications into fault-tolerant, distributed implementations, called *resilient objects* [Birman-a]. Resilient objects achieve fault tolerance by replicating the code and data managed by the object at more than one site. The resulting *components* synchronize their actions to provide the effect of a single-site object. In the presence of failures, any ongoing operation at a failed component is continued by an operational one. Also, a resilient object continues to accept and process new operations as long as at least one component is operational. Finally, failed com-

ponents recover automatically when the site at which they reside is restarted.

The initial version of *ISIS* used a simple communication layer that provided an atomic broadcast with no ordering properties. This was unsatisfactory for two reasons. First, the implementation grew very complex because of the need to include, in various parts of the system, protocols to preclude orderings that might lead to inconsistencies, especially in the presence of failures. Second, the high degree of synchronization resulting from these protocols lowered system performance. When reimplemented using a preliminary version of the primitives presented here, the system became much simpler, and performance improved due to the highly concurrent nature of the primitives. Below, we describe some areas in which these benefits were obtained.

5.1. Updating replicated data

When replicated data are updated, care must be taken to ensure that the updates occur in the same order at all copies. Otherwise, the copies can become inconsistent. In an environment where no ordering properties are guaranteed on broadcasts, this is done by preceding an update to a local copy by a broadcast to the remote copies and waiting for confirmation from the remote copies that the update has been carried out, before allowing another local update to occur. This kind of synchronization means that the rate at which updates can occur is limited by the time it takes for a message to travel a round trip, which can be unacceptably high. If *OBCAST*'s are instead used to instruct remote copies to perform updates, an update can be considered complete when the local update is carried out. No further synchronization is required, because the properties of *OBCAST*'s guarantee that all the copies receive the update, and do so in the required order. The rate at which updates can now be performed is now the rate at which local updates can be done, which is usually much higher than the previous case. At the same time, the protocol for carrying out a replicated update is much simpler, as it consists of a single *OBCAST*. An efficient way of labeling such *OBCASTS* is described in Sec. 5.4.

5.2. Coordinator-cohort computations

In *ISIS*, one of the components of a resilient object is designated as the *coordinator* for the execution of a particular operation. The others, its *cohorts*, act as passive backups. If the coordinator fails, a cohort takes over as and restarts the request (details are given in [Birman-b] and [Birman-c]).

The process group abstraction facilitates the implementation of coordinator-cohort computations. The components of a resilient object is placed in the same process group, and each request to perform an operation is *OBCAST* to all the components. Since each component has the same process group view, they can independently decide on a unique coordinator for the request by using the same algorithm, without running an agreement protocol[5].

The *GBCAST* ordering properties prevent inconsistencies from arising when failures or recoveries occur. After a failure, cohorts can pick a new coordinator consistently, and because all have received the same messages from the previous coordinator, the object data is in a consistent state at all components. When a component recovers, it uses *GBCAST* to rejoin the group, hence all the operational components receive the *GBCAST* in the same state and any can transfer data to reinitialize the recovering component.

5.3. Managing locks on replicated data

Lock-based concurrency control is the most common method for obtaining serializability [Bernstein]. The usual locking method for replicated data is to obtain write locks on all copies and read locks on only one. This means that if the site at which a read lock is obtained were to fail, all information about this read lock would be lost. Hence if the recovery scheme requires that the reads and writes be serialized in the same way after a failure as before, information about read locks must be replicated. In the *ISIS* recovery scheme, as in many that use a saved state for recovery, it is necessary for the executions to be deterministic, and a change in serialization order

[5] In *ISIS* this is done as follows. If a request arrives from site s, the coordinator is the site t in the process group view minimizing $abs(t - s)$. This tends to locate the coordinator for a computation at a same site as the site where the request originated, which in *ISIS* improves response time.

would violate this. However, acquiring read locks at all sites would be inefficient. Instead, the ordering properties of the broadcast primitives are used to obtain the same effect.

A read-lock is first obtained locally. Then, a *read lock registration* message is *OBCAST* to the other copies of the data item. The sender immediately continues execution, as if its read-lock were already replicated, although the message may not actually have been delivered anywhere. If the sender fails before any message leaves the site, the effect is as if the read never occurred (recall that a failure destroys all information at a site). If, on the other hand, a site has received any message *m* sent after the lock acquisition, the *GBCAST* protocol for the failure will ensure that the read-lock registration message is delivered before the failure is detected by the process managing the lock. Thus, the read-lock behaves like a fully replicated one.

Unlike a read-lock, a write-lock must be explicitly granted by all components of an object. However, a deadlock could occur if concurrent write-lock requests on the same data item are granted in different orders by different components. This problem can be avoided by using *BCAST*'s for write lock acquisition requests. If the data item name is used as a *BCAST* label, write lock requests on the same data item are ordered in the same way at all components, and deadlock is avoided.

5.4. Performance issues

A prototype communication layer similar to the one described here has been operation since Jan. 1985 [Birman-c], and is being reimplemented to correspond exactly. Two performance measures are of interest. One, the response time for a typical request, measures the critical path before a reply can be issued to a caller. We considered a fault-tolerant object implemented using *ISIS* and distributed to 3 sites (SUN workstations). A request that acquires a replicated write-lock, updates a replicated data item, and then responds to its caller sends its reply after about .6 seconds; additional updates delay the response by .1 seconds each (the difference reflects the one-time cost of concurrency control). When *ISIS* is run in a synchronous mode, verifying that each update has actually completed before the coordinator undertakes any subsequent operations, such

a computation requires 1.5 seconds, with additional updates requiring .5 seconds each. Moreover, the performance of the synchronous version degrades as the number of sites increases, while the concurrent version gives the same performance regardless of the number of participating sites. Thus, concurrent communication primitives can have a substantial impact on performance.

A computation can remain active long after replying to the process that initiated it if a high level of concurrency is achieved. To isolate the effect of concurrency on the above figures, the total elapsed time between the issuing of the request and the true termination of the operation can be measured. In *ISIS*, we find that a single asynchronous update terminates after about 1 second, with subsequent updates delaying termination by about .3 seconds each, and with linear degradation as the number of sites increases.

The delay to termination would not be an issue unless computations at different sites compete for a lock, which should be rare in *ISIS*. Thus, for *ISIS* and for many other applications, the communication primitives described in this paper permit extremely good performance -- almost as good as for a non-distributed system performing the same operations -- but with the ability to tolerate failure as well.

6. Future Work

A weakness of the work described in this paper is its inability to permit continued operation in the presense of partitioning failures, when two or more subgroups of operational sites form, within which communication remains possible, but between which it is degraded or impossible. We are now investigating the adaptation of methods from [ElAbbadi-a] [ElAbbadi-b] to address this issue. Also interesting is the possibility of integrating communication primitives with synchronized clocks, for use in shared memory systems and tightly coupled multi-processors.

7. Conclusions

Our experience implementing a substantial fault-tolerant system lead to insights into the properties desired from a communication subsystem. The primitives described in this paper present a

simple interface, achieve a high level of concurrency, and are applicable to software ranging from distributed database systems to the fault-tolerant objects provided by *ISIS*. By respecting desired event orderings, they introduce a desirable form of determinism into distributed computation. A consequence is that high-level algorithms are greatly simplified, reducing the probability of error. We believe that this is a very promising and practical approach to building large fault-tolerant distributed systems, and the only one leading to confidence in the correctness of the resulting software.

8. Acknowledgments

Particular thanks go to Amr El Abbadi, Ozalp Babaoglu, and Thomas Raeuchle for their many comments. We are also indebted to Jay Misra and Mani Chandy for discussions and comments about an early draft of this paper, and to Dale Skeen, who helped found the *ISIS* group in 1982, and was responsible for the ordering algorithm used in the *BCAST* protocol.

9. References

[Bernstein] Bernstein, P., Goodman, N. Concurrency control algorithms for replicated database systems. *ACM Computing Surveys* 13, 2 (June 1981), 185-222.

[Birman-a] Birman, K., Dietrich, W., EL Abbadi, A., Joseph, T., Raeuchle, T. An overview of the *ISIS* project. *Newsletter of the IEEE Special Interest Group on Distributed Computing*. June 1985.

[Birman-b] Birman, K., *et. al.* Implementing fault-tolerant distributed objects. *IEEE TSE-11*, 6, (June 1985), 502-508.

[Birman-c] Birman, K. Replication and availability in the *ISIS* system. Dept. of Computer Science, Cornell Univ., TR 85-668, March 1985. *To appear: ACM 10th SOSP.*

[Chang] Chang, J, Maxemchuk, N. Reliable broadcast protocols. *ACM TOCS* 2, 3 (Aug. 1984), 251-273.

[Cooper] Cooper, E. Replicated distributed programs. Ph.D. dissertation, Computer Science Department, Univ. of California, Berkeley (May 1985).

[Cristian] Cristian, F., *et al.* Atomic broadcast: From simple message diffusion to Byzantine agreement. IBM Technical Report RJ 4540 (48668), Oct. 1984.

[ElAbbadi-a] El Abbadi, A., Skeen, D., Cristian, F. An efficient algorithm for replicated data management. *Proc. PODS*, Portland, OR, March 1985.

[ElAbbadi-b] El Abbadi, A., Toueg, S. Handling partitioning in distributed database systems. Forthcoming.

[Goodman] Goodman, N., *et al.* A recovery algorithm for a distributed database system. *Proc 2nd ACM PODS*, Atlanta, GA (March 1983), 8-15.

[Lamport] Time, clocks, and the ordering of events in a distributed system. *CACM* 21, 7, July 1978, 558-565.

[Schlicting] Schlicting, R, Schneider, F. Fail-stop processors: An approach to designing fault-tolerant distributed computing systems. *ACM TOCS* 1, 3, August 1983, 222-238.

[Skeen-a] Skeen, D. Crash recovery in distributed database systems. Ph.D. dissertation, Department of EECS, U.C. Berkeley, 1980.

[Skeen-b] Skeen, D. Determining the last process to fail. *ACM TOCS* 3, 1, Feb. 1985, 15-30.

[Skeen-c] Skeen, D. Unpublished communication.